Guidelines for Writing FPU Exception Handlers 35

As described in “Floating-Point Unit”, the Intel Architecture supports two mechanisms for accessing exception handlers to handle unmasked FPU exceptions: native mode and MS-DOS compatibility mode. The primary purpose of this appendix is to provide detailed information to help software engineers design and write FPU exception-handling facilities to run on PC systems that use the MS-DOS compatibility mode[1] for handling FPU exceptions. Some of the information in this appendix will also be of interest to engineers who are writing native-mode FPU exception handlers. The information provided is as follows:

- Discussion of the origin of the MS-DOS* FPU exception handling mechanism and its relationship to the FPU’s native exception handling mechanism.
- Description of the Intel Architecture flags and processor pins that control the MS-DOS FPU exception handling mechanism.
- Description of the external hardware typically required to support MS-DOS exception handling mechanism.
- Description of the FPU’s exception handling mechanism and the typical protocol for FPU exception handlers.
- Code examples that demonstrate various levels of FPU exception handlers.
- Discussion of FPU considerations in multitasking environments.
- Discussion of native mode FPU exception handling.

The information given is oriented toward the most recent generations of Intel architecture processors, starting with the Intel486. It is intended to augment the reference information given in Chapter 7, *Floating-Point Unit*.

A more extensive version of this appendix is available in the application note AP-578, *Software and Hardware Considerations for FPU Exception Handlers for Intel Architecture Processors* (Order Number 242415-001), which is available from Intel.

35.1 Origin of the MS-DOS* Compatibility Mode for Handling FPU Exceptions

The first generations of Intel Architecture processors (starting with the Intel 8086 and 8088 processors and going through the Intel 286 and Intel386 processors) did not have an on-chip floating-point unit. Instead, floating-point capability was provided on a separate numeric coprocessor chip. The first of these numeric coprocessors was the Intel 8087, which was followed by the Intel 287 and Intel 387 numeric coprocessors.

1. Microsoft Windows* 95 and Windows* 3.1 (and earlier versions) operating systems use almost the same FPU exception handling interface as the operating system. The recommendations in this appendix for a MS-DOS* compatible exception handler thus apply to all three operating systems.

To allow the 8087 to signal floating-point exceptions to its companion 8086 or 8088, the 8087 has an output pin, INT, which it asserts when an unmasked floating-point exception occurs. The designers of the 8087 recommended that the output from this pin be routed through a programmable interrupt controller (PIC) such as the Intel 8259A to the INTR pin of the 8086 or 8088. The accompanying interrupt vector number could then be used to access the floating-point exception handler.

However, the original IBM PC design and MS-DOS operating system used a different mechanism for handling the INT output from the 8087. It connected the INT pin directly to the NMI input pin of the 8086 or 8088. The NMI interrupt handler then had to determine if the interrupt was caused by a floating-point exception or another NMI event. This mechanism is the origin of want is now called the "MS-DOS compatibility mode." The decision to use this latter floating-point exception handling mechanism came about because when the IBM PC was first designed, the 8087 was not available. When the 8087 did become available, other functions had already been assigned to the eight inputs to the PIC. One of these functions was a BIOS video interrupt, which was assigned to interrupt number 16 for the 8086 and 8088.

The Intel 286 processor created the "native mode" for handling floating-point exceptions by providing a dedicated input pin (ERROR#) for receiving floating-point exception signals and a dedicated interrupt number, 16. Interrupt 16 was used to signal floating-point errors (also called math faults). It was intended that the ERROR# pin on the Intel 286 be connected to a corresponding ERROR# pin on the Intel 287 numeric coprocessor. When the Intel 287 signals a floating-point exception using this mechanism, the Intel 286 generates an interrupt 16, to invoke the floating-point exception handler.

To maintain compatibility existing PC software, the native floating-point exception handling mode of the Intel 286 and 287 was not used in the IBM PC AT* system design. Instead, the ERROR# pin on the Intel 286 was tied permanently high, and the ERROR# pin from the Intel 287 was routed to a second (cascaded) PIC. The resulting output of this PIC was routed through an exception handler and eventually caused an interrupt 2 (NMI interrupt). Here the NMI interrupt was shared with PC AT's new parity checking feature. Interrupt 16 remained assigned to the BIOS video interrupt handler. The external hardware for the MS-DOS compatibility mode must prevent the Intel 286 processor from executing past the next FPU instruction when an unmasked exception has been generated. To do this, it asserts the BUSY# signal into the Intel 286 when the ERROR# signal is asserted by the Intel 287.

The Intel386 processor and its companion Intel 387 numeric coprocessor provided the same hardware mechanism for signaling and handling floating-point exceptions as the Intel 286 and 287 processors. And again, to maintain compatibility with existing MS-DOS software, basically the same MS-DOS compatibility floating-point exception handling mechanism that was used in the PC AT was used in PCs based on the Intel386.

35.2 Implementation of the MS-DOS* Compatibility Mode In the Intel486™, Pentium®, and Pentium Pro Processors

Beginning with the Intel486 processor, the Intel Architecture provided a dedicated mechanism for enabling the MS-DOS compatibility mode for FPU exceptions and for generating external FPU-exception signals while operating in this mode. The following sections describe the implementation of the MS-DOS compatibility mode in Intel486 and Pentium processors and in the Pentium Pro processor. Also described is the recommended external hardware to support this mode of operation.

35.2.1 MS-DOS* Compatibility Mode in the Intel486™ and Pentium® Processors

In the Intel486, several things were done to enhance and speed up the numeric coprocessor, now called the floating-point unit (FPU). The most important enhancement was that the FPU was included in the same chip as the processor, for increased speed in FPU computations and reduced latency for FPU exception handling. Also, for the first time, the MS-DOS compatibility mode was built into the chip design, with the addition of the NE bit in control register CR0 and the addition of the FERR# (Floating point ERRor) and IGNNE# (IGNore Numeric Error) pins.

The NE bit selects the native FPU exception handling mode (NE = 1) or the MS-DOS compatibility mode (NE = 0). When native mode is selected, all signaling of floating-point exceptions is handled internally in the Intel486 chip, resulting in the generation of an interrupt 16.

When MS-DOS compatibility mode is selected the FERRR# and IGNNE# pins are used to signal floating-point exceptions. The FERR# output pin, which replaces the ERROR# pin from the previous generations of Intel Architecture numeric coprocessors, is connected to a PIC. A new input signal, IGNNE#, is provided to allow the FPU exception handler to execute FPU instructions, if desired, without first clearing the error condition and without triggering the interrupt a second time. This IGNNE# feature is needed to replicate the capability that was provided on MS-DOS compatibility Intel 286 and Intel 287 and Intel386 and Intel 387 systems by turning off the BUSY# signal, when inside the FPU exception handler, before clearing the error condition.

Note that Intel, in order to provide Intel486 processors for market segments which had no need for an FPU, created the “SX” versions. These Intel486 SX processors did not contain the floating point unit. Intel also produced Intel 487 SX processors for end users who later decided to upgrade to a system with an FPU. These Intel 487 SX processors are similar to standard Intel486 processors with a working FPU on board. Thus, the external circuitry necessary to support the MS-DOS compatibility mode for Intel 487 SX processors is the same as for standard Intel486 DX processors.

The Pentium and Pentium Pro processors offer the same mechanism (the NE bit and the FERR# and IGNNE# pins) as the Intel486 processors for generating FPU exceptions in MS-DOS compatibility mode. The actions of these mechanisms are slightly different and more straightforward for the Pentium Pro processors, as described in “MS-DOS* Compatibility Mode in the Pentium® Pro Processor”.

For Pentium and Pentium Pro processors, it is important to note that the special DP (Dual Processing) mode for Pentium Processors and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatibility FPU mode for systems using more than one processor.

35.2.1.1 Basic Rules: When FERR# Is Generated

When MS-DOS compatibility mode is enabled for the Intel486 or Pentium processors (NE bit is set to 0) and the IGNNE# input pin is de-asserted, the FERR# signal is generated as follows:

1. When an FPU instruction causes an unmasked FPU exception, the processor (in most cases) uses a “deferred” method of reporting the error. This means that the processor does not respond immediately, but rather freezes just before executing the next WAIT or FPU instruction (except for “no-wait” instructions, which the FPU executes regardless of an error condition).
2. When the processor freezes, it also asserts the FERR# output.

3. The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# assertion.
4. In MS-DOS* compatibility systems, FERR# is fed to the IRQ13 input in the cascaded PIC. The PIC generates interrupt 75H, which then branches to interrupt 2, as described earlier in this appendix for systems using the Intel 286 and Intel 287 or Intel386 and Intel 387 processors.

The deferred method of error reporting is used for all exceptions caused by the basic arithmetic instructions (including FADD, FSUB, FMUL, FDIV, FSQRT, FCOM and FUCOM), for precision exceptions caused by all types of FPU instructions, and for numeric underflow and overflow exceptions caused by all types of FPU instructions except stores to memory.

Some FPU instructions with some FPU exceptions use an “immediate” method of reporting errors. Here, the FERR# is asserted immediately, at the time that the exception occurs. The immediate method of error reporting is used for FPU stack fault, invalid operation and denormal exceptions caused by all transcendental instructions, FSCALE, FXTRACT, FPREM and others, and all exceptions (except precision) when caused by FPU store instructions. Like deferred error reporting, immediate error reporting will cause the processor to freeze just before executing the next WAIT or FPU instruction if the error condition has not been cleared by that time.

Note that in general, whether deferred or immediate error reporting is used for an FPU exception depends both on which exception occurred and which instruction caused that exception. A complete specification of these cases, which applies to both the Pentium and the Intel486 processors, is given in Section 5.1.2 in the *Pentium® Processor Family Developer’s Manual: Volume 1*.

If NE=0 but the IGNNE# input is active while an unmasked FPU exception is in effect, the processor disregards the exception, does not assert FERR#, and continues. If IGNNE# is then de-asserted and the FPU exception has not been cleared, the processor will respond as described above. (That is, an immediate exception case will assert FERR# immediately. A deferred exception case will assert FERR# and freeze just before the next FPU or WAIT instruction.) The assertion of IGNNE# is intended for use only inside the FPU exception handler, where it is needed if one wants to execute non-control FPU instructions for diagnosis, before clearing the exception condition. When IGNNE# is asserted inside the exception handler, a preceding FPU exception has already caused FERR# to be asserted, and the external interrupt hardware has responded, but IGNNE# assertion still prevents the freeze at FPU instructions. Note that if IGNNE# is left active outside of the FPU exception handler, additional FPU instructions may be executed after a given instruction has caused an FPU exception. In this case, if the FPU exception handler ever did get invoked, it could not determine which instruction caused the exception.

To properly manage the interface between the processor’s FERR# output, its IGNNE# input, and the IRQ13 input of the PIC, additional external hardware is needed. A recommended configuration is described in the following section.

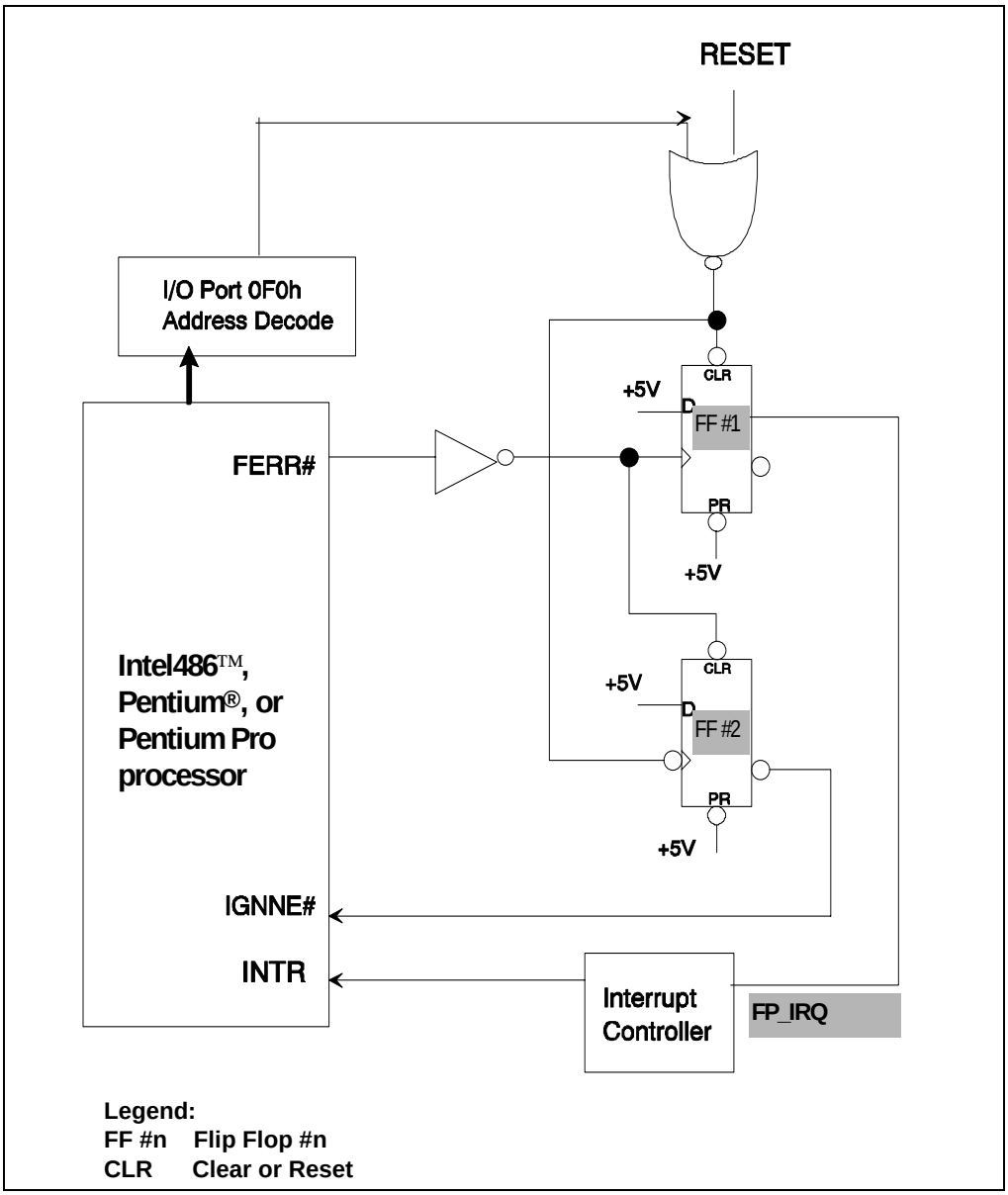
35.2.1.2 Recommended External Hardware to Support the MS-DOS* Compatibility Mode

Figure 35-1 provides an external circuit that will assure proper handling of FERR# and IGNNE# when an FPU exception occurs. In particular, it assures that IGNNE# will be active only inside the FPU exception handler without depending on the order of actions by the exception handler. Some hardware implementations have been less robust because they have depended on the exception handler to clear the FPU exception interrupt request to the PIC (FP_IRQ signal) **before** the handler

causes FERR# to be de-asserted by clearing the exception from the FPU itself. Figure 35-2 shows the details of how IGNNE# will behave when the circuit in Figure 35-1 is implemented. The temporal regions within the FPU exception handler activity are described as follows:

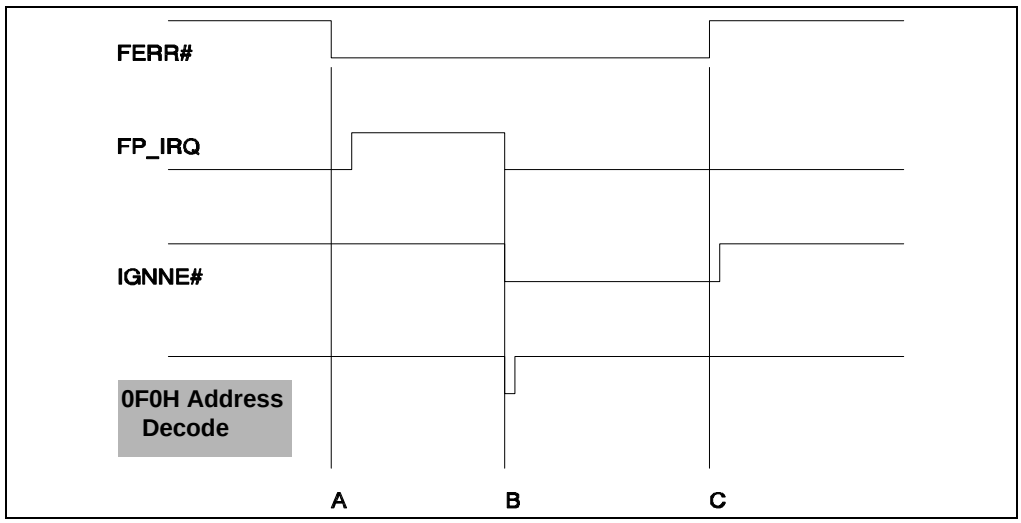
1. The FERR# signal is activated by an FPU exception and sends an interrupt request through the PIC to the processor's INTR pin.
2. During the FPU interrupt service routine (exception handler) the processor will need to clear the interrupt request latch (Flip Flop #1). It may also want to execute non-control FPU instructions before the exception is cleared from the FPU. For this purpose the IGNNE# must be driven low. Typically in the PC environment an I/O access to Port 0F0H clears the external FPU exception interrupt request (FP_IRQ). In the recommended circuit, this access also is used to activate IGNNE#. With IGNNE# active the FPU exception handler may execute any FPU instruction without being blocked by an active FPU exception.
3. Clearing the exception within the FPU will cause the FERR# signal to be deactivated and then there is no further need for IGNNE# to be active. In the recommended circuit, the deactivation of FERR# is used to deactivate IGNNE#. If another circuit is used, the software and circuit together must assure that IGNNE# is deactivated no later than the exit from the FPU exception handler.

Figure 35-1. Recommended Circuit for MS-DOS* Compatibility FPU Exception Handling



In the circuit in Figure 35-1, when the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. So the handler can activate IGNNE#, if needed, by doing this 0F0H access before clearing the FPU exception condition (which de-asserts FERR#). However, the circuit does not depend on the order of actions by the FPU exception handler to guarantee the correct hardware state upon exit from the handler. Flip Flop #2, which drives IGNNE# to the processor, has its CLEAR input attached to the inverted FERR#. This ensures that IGNNE# can never be active when FERR# is inactive. So if the handler clears the FPU exception condition **before** the 0F0H access, IGNNE# does not get activated and left on after exit from the handler.

Figure 35-2. Behavior of Signals During FPU Exception Handling



35.2.1.3 No-Wait FPU Instructions Can Get FPU Interrupt in Window

The Pentium and Intel486 processors implement the “no-wait” floating-point instructions (FNINIT, FNCLEX, FNSTENV, FNSAVE, FNSTSW, FNSTCW, FNENI, FNDISI or FNSETPM) in the MS-DOS compatibility mode in the following manner. (See “FPU Control Instructions” and “Waiting Vs. Non-waiting Instructions” for a discussion of the no-wait instructions.)

If an unmasked numeric exception is pending from a preceding FPU instruction, a member of the no-wait class of instructions will, at the beginning of its execution, assert the FERR# pin in response to that exception just like other FPU instructions, but then, unlike the other FPU instructions, FERR# will be de-asserted. This de-assertion was implemented to allow the no-wait class of instructions to proceed without an interrupt due to any pending numeric exception. However, the brief assertion of FERR# is sufficient to latch the FPU exception request into most hardware interface implementations (including Intel’s recommended circuit).

All the FPU instructions are implemented such that during their execution, there is a window in which the processor will sample and accept external interrupts. If there is a pending interrupt, the processor services the interrupt first before resuming the execution of the instruction. Consequently, it is possible that the no-wait floating-point instruction may accept the external interrupt caused by it’s own assertion of the FERR# pin in the event of a pending unmasked numeric exception, which is not an explicitly documented behavior of a no-wait instruction. This process is illustrated in Figure 35-3.

Figure 35-3. Timing of Receipt of External Interrupt

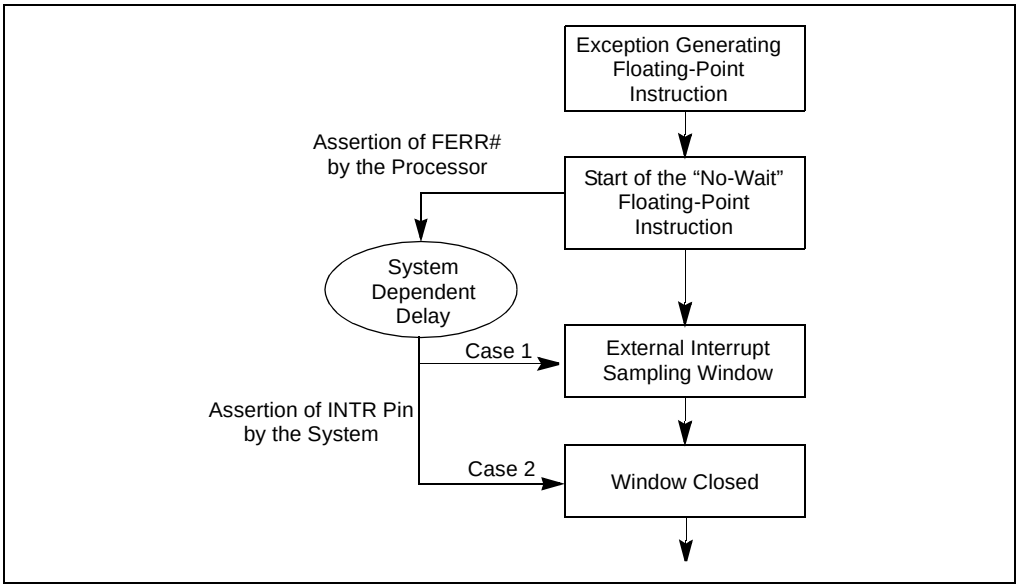


Figure 35-3 assumes that a floating-point instruction that generates a "deferred" error (as defined in the "Basic Rules: When FERR# Is Generated"), which asserts the FERR# pin only on encountering the next floating-point instruction, causes an unmasked numeric exception. Assume that the next floating-point instruction following this instruction is one of the no-wait floating-point instructions. The FERR# pin is asserted by the processor to indicate the pending exception on encountering the no-wait floating-point instruction. After the assertion of the FERR# pin the no-wait floating-point instruction opens a window where the pending external interrupts are sampled.

Then there are two cases possible depending on the timing of the receipt of the interrupt via the INTR pin (asserted by the system in response to the FERR# pin) by the processor.

Case 1 — If the system responds to the assertion of FERR# pin by the no-wait floating-point instruction via the INTR pin during this window then the interrupt is serviced first, before resuming the execution of the no-wait floating-point instruction.

Case 2 — If the system responds via the INTR pin after the window has closed then the interrupt is recognized only at the next instruction boundary.

There are two other ways, in addition to Case 1 above, in which a no-wait floating-point instruction can service a numeric exception inside its interrupt window. First, the first floating-point error condition could be of the "immediate" category (as defined in "Basic Rules: When FERR# Is Generated") that asserts FERR# immediately. If the system delay before asserting INTR is long enough, relative to the time elapsed before the no-wait floating-point instruction, INTR can be asserted inside the interrupt window for the latter. Second, consider two no-wait FPU instructions in close sequence, and assume that a previous FPU instruction has caused an unmasked numeric exception. Then if the INTR timing is too long for an FERR# signal triggered by the first no-wait instruction to hit the first instruction's interrupt window, it could catch the interrupt window of the second.

The possible malfunction of a no-wait FPU instruction explained above cannot happen if the instruction is being used in the manner for which Intel originally designed it. The no-wait instructions were intended to be used inside the FPU exception handler, to allow manipulation of

the FPU before the error condition is cleared, without hanging the processor because of the FPU error condition, and without the need to assert IGNNE#. They will perform this function correctly, since before the error condition is cleared, the assertion of FERR# that caused the FPU error handler to be invoked is still active. Thus the logic that would assert FERR# briefly at a no-wait instruction causes no change since FERR# is already asserted. The no-wait instructions may also be used without problem in the handler after the error condition is cleared, since now they will not cause FERR# to be asserted at all.

If a no-wait instruction is used outside of the FPU exception handler, it may malfunction as explained above, depending on the details of the hardware interface implementation and which particular processor is involved. The actual interrupt inside the window in the no-wait instruction may be blocked by surrounding it with the instructions: PUSHFD, CLI, no-wait, then POPFD. (CLI blocks interrupts, and the push and pop of flags preserves and restores the original value of the interrupt flag.) However, if FERR# was triggered by the no-wait, its latched value and the PIC response will still be in effect. Further code can be used to check for and correct such a condition, if needed. “Considerations When FPU Shared Between Tasks”, discusses an important example of this type of problem and gives a solution.

35.2.2 MS-DOS* Compatibility Mode in the Pentium® Pro Processor

When bit NE=0 in CR0, the MS-DOS compatibility mode of the Pentium Pro processor provides FERR# and IGNNE# functionality that is almost identical to the Intel486 and Pentium processors. The same external hardware described in “Recommended External Hardware to Support the MS-DOS* Compatibility Mode”, is recommended for the Pentium Pro processor as well as the two previous generations. The only change to MS-DOS compatibility FPU exception handling with the Pentium Pro processor is that all exceptions for all FPU instructions cause immediate error reporting. That is, FERR# is asserted as soon as the FPU detects an unmasked exception; there are no cases in which error reporting is deferred to the next FPU or WAIT instruction. (As is discussed in “Basic Rules: When FERR# Is Generated”, most exception cases in the Intel486 and Pentium processors are of the deferred type.)

Although FERR# is asserted immediately upon detection of an unmasked FPU error, this certainly does not mean that the requested interrupt will always be serviced before the next instruction in the code sequence is executed. To begin with, the Pentium Pro processor executes several instructions simultaneously. There also will be a delay, which depends on the external hardware implementation, between the FERR# assertion from the processor and the responding INTR assertion to the processor. Further, the interrupt request to the PICs (IRQ13) may be temporarily blocked by the operating system, or delayed by higher priority interrupts, and processor response to INTR itself is blocked if the operating system has cleared the IF bit in EFLAGS.

However, just as with the Intel486 and Pentium processors, if the IGNNE# input is inactive, a floating point exception which occurred in the previous FPU instruction and is unmasked causes the processor to freeze immediately when encountering the next WAIT or FPU instruction (except for no-wait instructions). This means that if the FPU exception handler has not already been invoked due to the earlier exception (and therefore, the handler not has cleared that exception state from the FPU), the processor is forced to wait for the handler to be invoked and handle the exception, before the processor can execute another WAIT or FPU instruction.

As explained in “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, if a no-wait instruction is used outside of the FPU exception handler, in the Intel486 and Pentium processors, it may accept an unmasked exception from a previous FPU instruction which happens to fall within

the external interrupt sampling window that is opened near the beginning of execution of all FPU instructions. This will not happen in the Pentium Pro processor, because this sampling window has been removed from the no-wait group of FPU instructions.

35.3 Recommended Protocol for MS-DOS* Compatibility Handlers

The activities of numeric programs can be split into two major areas: program control and arithmetic. The program control part performs activities such as deciding what functions to perform, calculating addresses of numeric operands, and loop control. The arithmetic part simply adds, subtracts, multiplies, and performs other operations on the numeric operands. The processor is designed to handle these two parts separately and efficiently. An FPU exception handler, if a system chooses to implement one, is often one of the most complicated parts of the program control code.

35.3.1 Floating-Point Exceptions and Their Defaults

The FPU can recognize six classes of floating-point exception conditions while executing floating-point instructions:

1. #I — Invalid operation
   #IS — Stack fault
   #IA — IEEE standard invalid operation
2. #Z — Divide-by-zero
3. #D — Denormalized operand
4. #O — Numeric overflow
5. #U — Numeric underflow
6. #P — Inexact result (precision)

For complete details on these exceptions and their defaults, see “Floating-Point Exception Handling” and “Floating-Point Exception Conditions”.

35.3.2 Two Options for Handling Numeric Exceptions

Depending on options determined by the software system designer, the processor takes one of two possible courses of action when a numeric exception occurs:

- The FPU can handle selected exceptions itself, producing a default fix-up that is reasonable in most situations. This allows the numeric program execution to continue undisturbed. Programs can mask individual exception types to indicate that the FPU should generate this safe, reasonable result whenever the exception occurs. The default exception fix-up activity is treated by the FPU as part of the instruction causing the exception; no external indication of the exception is given (except that the instruction takes longer to execute when it handles a masked exception.) When masked exceptions are detected, a flag is set in the numeric status register, but no information is preserved regarding where or when it was set.
- Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. The exception

handler can then implement any sort of recovery procedures desired for any numeric exception detectable by the FPU.

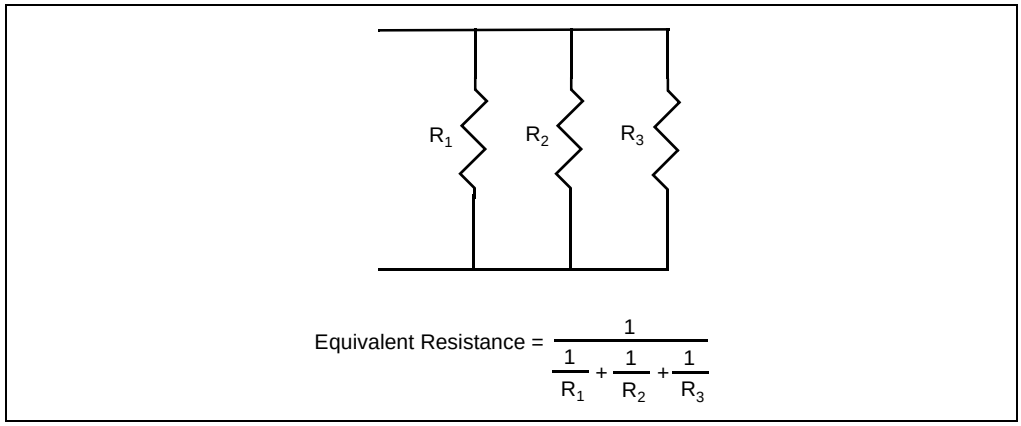
35.3.2.1 Automatic Exception Handling: Using Masked Exceptions

Each of the six exception conditions described above has a corresponding flag bit in the FPU status word and a mask bit in the FPU control word. If an exception is masked (the corresponding mask bit in the control word = 1), the processor takes an appropriate default action and continues with the computation. The processor has a default fix-up activity for every possible exception condition it may encounter. These masked-exception responses are designed to be safe and are generally acceptable for most numeric applications.

For example, if the Inexact result (Precision) exception is masked, the system can specify whether the FPU should handle a result that cannot be represented exactly by one of four modes of rounding: rounding it normally, chopping it toward zero, always rounding it up, or always down. If the Underflow exception is masked, the FPU will store a number that is too small to be represented in normalized form as a denormal (or zero if it's smaller than the smallest denormal). Note that when exceptions are masked, the FPU may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the FPU could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow.

As an example of how even severe exceptions can be handled safely and automatically using the default exception responses, consider a calculation of the parallel resistance of several values using only the standard formula (see Figure 35-4). If R1 becomes zero, the circuit resistance becomes zero. With the divide-by-zero and precision exceptions masked, the processor will produce the correct result. FDIV of R1 into 1 gives infinity, and then FDIV of (infinity +R2 +R3) into 1 gives zero.

Figure 35-4. Arithmetic Example Using Infinity



By masking or unmasking specific numeric exceptions in the FPU control word, programmers can delegate responsibility for most exceptions to the processor, reserving the most severe exceptions for programmed exception handlers. Exception-handling software is often difficult to write, and the masked responses have been tailored to deliver the most reasonable result for each condition. For the majority of applications, masking all exceptions yields satisfactory results with the least programming effort. Certain exceptions can usefully be left unmasked during the debugging phase of software development, and then masked when the clean software is actually run. An invalid-operation exception for example, typically indicates a program error that must be corrected.

The exception flags in the FPU status word provide a cumulative record of exceptions that have occurred since these flags were last cleared. Once set, these flags can be cleared only by executing the FCLEX/FNCLEX (clear exceptions) instruction, by reinitializing the FPU with FINIT/FNINIT or FSAVE/FNSAVE, or by overwriting the flags with an FRSTOR or FLDENV instruction. This allows a programmer to mask all exceptions, run a calculation, and then inspect the status word to see if any exceptions were detected at any point in the calculation.

35.3.2.2 Software Exception Handling

If the FPU in or with an Intel Architecture processor (Intel 286 and onwards) encounters an unmasked exception condition, with the system operated in the MS-DOS compatibility mode and with IGNNE# not asserted, a software exception handler is invoked through a PIC and the processor's INTR pin. The FERR# (or ERROR#) output from the FPU that begins the process of invoking the exception handler may occur when the error condition is first detected, or when the processor encounters the next WAIT or FPU instruction. Which of these two cases occurs depends on the processor generation and also on which exception and which FPU instruction triggered it, as discussed earlier in "Origin of the MS-DOS* Compatibility Mode for Handling FPU Exceptions" and "Implementation of the MS-DOS* Compatibility Mode In the Intel486™, Pentium®, and Pentium Pro Processors". The elapsed time between the initial error signal and the invocation of the FPU exception handler depends of course on the external hardware interface, and also on whether the external interrupt for FPU errors is enabled. But the architecture ensures that the handler will be invoked before execution of the next WAIT or floating-point instruction since an unmasked floating-point exception causes the processor to freeze just before executing such an instruction (unless the IGNNE# input is active, or it is a no-wait FPU instruction).

The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# (or ERROR#) output of the processor (or coprocessor), usually through IRQ13 on the "slave" PIC, and then through INTR. Then the external interrupt invokes the exception handling routine. Note that if the external interrupt for FPU errors is disabled when the processor executes an FPU instruction, the processor will freeze until some other (enabled) interrupt occurs if an unmasked FPU exception condition is in effect. If NE = 0 but the IGNNE# input is active, the processor disregards the exception and continues. Error reporting via an external interrupt is supported for MS-DOS compatibility. Chapter 17, *Intel Architecture Compatibility* of the *Intel Architecture Software Developer's Manual, Volume 3*, contains further discussion of compatibility issues.

The references above to the ERROR# output from the FPU apply to the Intel 387 and Intel 287 math coprocessors (NPX chips). If one of these coprocessors encounters an unmasked exception condition, it signals the exception to the Intel 286 or Intel386 processor using the ERROR# status line between the processor and the coprocessor. See "Origin of the MS-DOS* Compatibility Mode for Handling FPU Exceptions", in this appendix, and Chapter 17, *Intel Architecture Compatibility*, in the *Intel Architecture Software Developer's Manual, Volume 3* for differences in FPU exception handling.

The exception-handling routine is normally a part of the systems software. The routine must clear (or disable) the active exception flags in the FPU status word before executing any floating point instructions that cannot complete execution when there is a pending floating point exception. Otherwise, the floating point instruction will trigger the FPU interrupt again, and the system will be caught in an endless loop of nested floating point exceptions, and hang. In any event, the routine must clear (or disable) the active exception flags in the FPU status word after handling them, and before IRET(D). Typical exception responses may include:

- Incrementing an exception counter for later display or printing.
- Printing or displaying diagnostic information (e.g., the FPU environment and registers).

- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it.

Applications programmers should consult their operating system's reference manuals for the appropriate system response to numerical exceptions. For systems programmers, some details on writing software exception handlers are provided in Chapter 5, *Interrupt and Exception Handling*, in the *Intel Architecture Software Developer's Manual, Volume 3*, as well as in "FPU Exception Handling Examples", in this appendix.

As discussed in "Recommended External Hardware to Support the MS-DOS* Compatibility Mode", some early FERR# to INTR hardware interface implementations are less robust than the recommended circuit. This is because they depended on the exception handler to clear the FPU exception interrupt request to the PIC (by accessing port 0F0H) **before** the handler causes FERR# to be de-asserted by clearing the exception from the FPU itself. To eliminate the chance of a problem with this early hardware, Intel recommends that FPU exception handlers always access port 0F0H before clearing the error condition from the FPU.

35.3.3 Synchronization Required for Use of FPU Exception Handlers

Concurrency or synchronization management requires a check for exceptions before letting the processor change a value just used by the FPU. It is important to remember that almost any numeric instruction can, under the wrong circumstances, produce a numeric exception.

35.3.3.1 Exception Synchronization: What, Why and When

Exception synchronization means that the exception handler inspects and deals with the exception in the context in which it occurred. If concurrent execution is allowed, the state of the processor when it recognizes the exception is often **not** in the context in which it occurred. The processor may have changed many of its internal registers and be executing a totally different program by the time the exception occurs. If the exception handler cannot recapture the original context, it cannot reliably determine the cause of the exception or to recover successfully from the exception. To handle this situation, the FPU has special registers updated at the start of each numeric instruction to describe the state of the numeric program when the failed instruction was attempted. This provides tools to help the exception handler recapture the original context, but the application code must also be written with synchronization in mind. Overall, exception synchronization must ensure that the FPU and other relevant parts of the context are in a well defined state when the handler is invoked after an unmasked numeric exception occurs.

When the FPU signals an unmasked exception condition, it is requesting help. The fact that the exception was unmasked indicates that further numeric program execution under the arithmetic and programming rules of the FPU will probably yield invalid results. Thus the exception must be handled, and with proper synchronization, or the program will not operate reliably.

For programmers in higher-level languages, all required synchronization is automatically provided by the appropriate compiler. However, for assembly language programmers exception synchronization remains the responsibility of the programmer. It is not uncommon for a programmer to expect that their numeric program will not cause numeric exceptions after it has been tested and debugged, but in a different system or numeric environment, exceptions may occur regularly nonetheless. An obvious example would be use of the program with some numbers beyond the range for which it was designed and tested. The example in "Exception Synchronization Examples", shows a more subtle way in which unexpected exceptions can occur.

As described in “Floating-Point Exceptions and Their Defaults”, depending on options determined by the software system designer, the processor can perform one of two possible courses of action when a numeric exception occurs.

- The FPU can provide a default fix-up for selected numeric exceptions. If the FPU performs its default action for all exceptions, then the need for exception synchronization is not manifest. However, code is often ported to contexts and operating systems for which it was not originally designed. The example below illustrates that it is safest to always consider exception synchronization when designing code that uses the FPU.
- Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. When an FPU exception handler will be invoked, synchronization must always be considered to assure reliable performance.

The following examples illustrate the need to always consider exception synchronization when writing numeric code, even when the code is initially intended for execution with exceptions masked.

35.3.3.2 Exception Synchronization Examples

In the following examples, three instructions are shown to load an integer, calculate its square root, then increment the integer. The synchronous execution of the FPU will allow both of these programs to execute correctly, with INC COUNT being executed in parallel in the processor, as long as no exceptions occur on the FILD instruction. However, if the code is later moved to an environment where exceptions are unmasked, the code in the first example will not work correctly:

Incorrect Error Synchronization

```
FILD    COUNT ; FPU instruction
INC     COUNT ; integer instruction alters operand
FSQRT         ; subsequent FPU instruction -- error
              ; from previous FPU instruction detected here
```

Proper Error Synchronization

```
FILD COUNT; FPU instruction
FSQRT   ; subsequent FPU instruction -- error from
        ; previous FPU instruction detected here
INC COUNT; integer instruction alters operand
```

In some operating systems supporting the FPU, the numeric register stack is extended to memory. To extend the FPU stack to memory, the invalid exception is unmasked. A push to a full register or pop from an empty register sets SF (Stack Fault flag) and causes an invalid operation exception. The recovery routine for the exception must recognize this situation, fix up the stack, then perform the original operation. The recovery routine will not work correctly in the first example shown in the figure. The problem is that the value of COUNT is incremented before the exception handler is invoked, so that the recovery routine will load an incorrect value of COUNT, causing the program to fail or behave unreliably.

35.3.3.3 Proper Exception Synchronization in General

As explained in “Recommended External Hardware to Support the MS-DOS* Compatibility Mode”, if the FPU encounters an unmasked exception condition a software exception handler is invoked **before** execution of the **next** WAIT or floating-point instruction. This is because an unmasked floating-point exception causes the processor to freeze immediately before executing such an instruction (unless the IGNNE# input is active, or it is a no-wait FPU instruction). Exactly when the exception handler will be invoked (in the interval between when the exception is detected and the next WAIT or FPU instruction) is dependent on the processor generation, the system, and which FPU instruction and exception is involved.

To be safe in exception synchronization, one should assume the handler will be invoked at the end of the interval. Thus the program should not change any value that might be needed by the handler (such as COUNT in the above example) until **after** the **next** FPU instruction following an FPU instruction that could cause an error. If the program needs to modify such a value before the next FPU instruction (or if the next FPU instruction could also cause an error), then a WAIT instruction should be inserted before the value is modified. This will force the handling of any exception before the value is modified. A WAIT instruction should also be placed after the last floating-point instruction in an application so that any unmasked exceptions will be serviced before the task completes.

35.3.4 FPU Exception Handling Examples

There are many approaches to writing exception handlers. One useful technique is to consider the exception handler procedure as consisting of “prologue,” “body,” and “epilogue” sections of code.

In the transfer of control to the exception handler due to an INTR, NMI, or SMI, external interrupts have been disabled by hardware. The prologue performs all functions that must be protected from possible interruption by higher-priority sources. Typically, this involves saving registers and transferring diagnostic information from the FPU to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler. The standard “prologue” not only saves the registers and transfers diagnostic information from the FPU to memory but also clears the floating point exception flags in the status word. Alternatively, when it is not necessary for the handler to be re-entrant, another technique may also be used. In this technique, the exception flags are not cleared in the “prologue” and the body of the handler must not contain any floating point instructions that cannot complete execution when there is a pending floating point exception. (The no-wait instructions are discussed in “Waiting Vs. Non-waiting Instructions”.) Note that the handler must still clear the exception flag(s) before executing the IRET. If the exception handler uses neither of these techniques the system will be caught in an endless loop of nested floating point exceptions, and hang.

The body of the exception handler examines the diagnostic information and makes a response that is necessarily application-dependent. This response may range from halting execution, to displaying a message, to attempting to repair the problem and proceed with normal execution. The epilogue essentially reverses the actions of the prologue, restoring the processor so that normal execution can be resumed. The epilogue must not load an unmasked exception flag into the FPU or another exception will be requested immediately.

The following code examples show the ASM386/486 coding of three skeleton exception handlers, with the save spaces given as correct for 32 bit protected mode. They show how prologues and epilogues can be written for various situations, but the application dependent exception handling body is just indicated by comments showing where it should be placed.

The first two are very similar; their only substantial difference is their choice of instructions to save and restore the FPU. The trade-off here is between the increased diagnostic information provided by FNSAVE and the faster execution of FNSTENV. (Also, after saving the original contents, FNSAVE re-initializes the FPU, while FNSTENV only masks all FPU exceptions.) For applications that are sensitive to interrupt latency or that do not need to examine register contents, FNSTENV reduces the duration of the "critical region," during which the processor does not recognize another interrupt request. (See the "Saving the FPU's State", for a complete description of the FPU save image.)

After the exception handler body, the epilogues prepare the processor to resume execution from the point of interruption (i.e., the instruction following the one that generated the unmasked exception). Notice that the exception flags in the memory image that is loaded into the FPU are cleared to zero prior to reloading (in fact, in these examples, the entire status word image is cleared).

Examples 35-1 and 35-2 assume that the exception handler itself will not cause an unmasked exception. Where this is a possibility, the general approach shown in Example 35-3 can be employed. The basic technique is to save the full FPU state and then to load a new control word in the prologue. Note that considerable care should be taken when designing an exception handler of this type to prevent the handler from being reentered endlessly.

Example 35-1. Full-State Exception Handler

```
SAVE_ALL PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
    PUSH EBP
    .
    .
    MOV  EBP, ESP
    SUB  ESP, 108 ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE FULL FPU STATE, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSAVE [EBP-108]
    PUSH [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
    POPFD   ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION
;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED STATE IMAGE
    MOV BYTE PTR [EBP-104], 0H
    FRSTOR  [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV  ESP, EBP
    .
    .
    POP  EBP
;
; RETURN TO INTERRUPTED CALCULATION
    IRETD
SAVE_ALL  ENDP
```

Example 35-2. Reduced-Latency Exception Handler

```
SAVE_ENVIRONMENTPROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU ENVIRONMENT
    PUSH  EBP
```

```
    .
    .
    MOV  EBP, ESP
    SUB  ESP, 28  ; ALLOCATES 28 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE ENVIRONMENT, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSTENV[EBP-28]
    PUSH  [EBP + OFFSET_TO_EFLAGS]  ; COPY OLD EFLAGS TO STACK TOP
    POPFD   ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED ENVIRONMENT IMAGE
    MOV  BYTE PTR [EBP-24], 0H
    FLDENV  [EBP-28]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV  ESP, EBP
    .
    .
    POP  EBP
;
; RETURN TO INTERRUPTED CALCULATION
    IRETD
SAVE_ENVIRONMENT ENDP
```

Example 35-3. Reentrant Exception Handle

```
    .
    .
LOCAL_CONTROL DW ? ; ASSUME INITIALIZED
    .
    .
REENTRANTPROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
    PUSH  EBP
    .
    .
    MOV  EBP, ESP
    SUB  ESP, 108  ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)

; SAVE STATE, LOAD NEW CONTROL WORD, RESTORE INTERRUPT ENABLE FLAG (IF)
    FNSAVE  [EBP-108]
    FLDCW  LOCAL_CONTROL
    PUSH  [EBP + OFFSET_TO_EFLAGS]  ; COPY OLD EFLAGS TO STACK TOP
    POPFD ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION.
    .
    .
;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE. AN UNMASKED EXCEPTION

; GENERATED HERE WILL CAUSE THE EXCEPTION HANDLER TO BE REENTERED.
; IF LOCAL STORAGE IS NEEDED, IT MUST BE ALLOCATED ON THE STACK.
;
    .
    .
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED STATE IMAGE
    MOV  BYTE PTR [EBP-104], 0H
    FRSTOR  [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
    MOV  ESP, EBP
    .
    .
    POP  EBP
```

```
;
; RETURN TO POINT OF INTERRUPTION
    IRETD
REENTRANT  ENDP
```

35.3.5 Need for Storing State of IGNNE# Circuit If Using FPU and SMM

The recommended circuit (see Figure 35-1) for MS-DOS compatibility FPU exception handling for Intel486 processors and beyond contains two flip flops. When the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. The assertion of IGNNE# may be used by the handler if needed to execute any FPU instruction while ignoring the pending FPU errors. The problem here is that the state of Flip Flop #2 is effectively an additional (but hidden) status bit that can affect processor behavior, and so ideally should be saved upon entering SMM, and restored before resuming to normal operation. If this is not done, and also the SMM code saves the FPU state, AND an FPU error handler is being used which relies on IGNNE# assertion, then (very rarely) the FPU handler will nest inside itself and malfunction. The following example shows how this can happen.

Suppose that the FPU exception handler includes the following sequence:

```
FNSTSW  save_sw ; save the FPU status word
        ; using a no-wait FPU instruction
OUT 0F0H, AL; clears IRQ13 & activates IGNNE#
    . . . .
FLDCW new_cw ; loads new CW ignoring FPU errors,
             ; since IGNNE# is assumed active; or any
             ; other FPU instruction that is not a no-wait
             ; type will cause the same problem
     . . . .
FCLEX   ; clear the FPU error conditions & thus turn off FERR# & reset the IGNNE# FF
```

The problem will only occur if the processor enters SMM between the OUT and the FLDCW instructions. But if that happens, AND the SMM code saves the FPU state using FNSAVE, then the IGNNE# Flip Flop will be cleared (because FNSAVE clears the FPU errors and thus de-asserts FERR#). When the processor returns from SMM it will restore the FPU state with FRSTOR, which will re-assert FERR#, but the IGNNE# Flip Flop will not get set. Then when the FPU error handler executes the FLDCW instruction, the active error condition will cause the processor to re-enter the FPU error handler from the beginning. This may cause the handler to malfunction.

To avoid this problem, Intel recommends two measures:

1. Do not use the FPU for calculations inside SMM code. (The normal power management, and sometimes security, functions provided by SMM have no need for FPU calculations; if they are needed for some special case, use scaling or emulation instead.) This eliminates the need to do FNSAVE/FRSTOR inside SMM code, except when going into a 0 V suspend state (in which, in order to save power, the CPU is turned off completely, requiring its complete state to be saved.)
2. The system should not call upon SMM code to put the processor into 0 V suspend while the processor is running FPU calculations, or just after an interrupt has occurred. Normal power management protocol avoids this by going into power down states only after timed intervals in which no system activity occurs.

35.3.6 Considerations When FPU Shared Between Tasks

The Intel Architecture allows speculative deferral of floating point state swaps on task switches. This feature allows postponing an FPU state swap until an FPU instruction is actually encountered in another task. Since kernel tasks rarely use floating point, and some applications do not use floating point or use it infrequently, the amount of time saved by avoiding unnecessary stores of the floating point state is significant. Speculative deferral of FPU saves does, however, place an extra burden on the kernel in three key ways:

1. The kernel must keep track of which thread owns the FPU, which may be different from the currently executing thread.
2. The kernel must associate any floating point exceptions with the generating task. This requires special handling since floating point exceptions are delivered asynchronous with other system activity.
3. There are conditions under which spurious floating point exception interrupts are generated, which the kernel must recognize and discard.

35.3.6.1 Speculatively Deferring FPU Saves, General Overview

In order to support multi-tasking, each thread in the system needs a save area for the general purpose registers, and each task that is allowed to use floating point needs an FPU save area large enough to hold the entire FPU stack and associated FPU state such as the control word and status word. (See “Saving the FPU’s State”, for a complete description of the FPU save image.)

On a task switch, the general purpose registers are swapped out to their save area for the suspending thread, and the registers of the resuming thread are loaded. The FPU state does not need to be saved at this point. If the resuming thread does not use the FPU before it is itself suspended, then both a save and a load of the FPU state has been avoided. It is often the case that several threads may be executed without any usage of the FPU.

The processor supports speculative deferral of FPU saves via interrupt 7 “Device Not Available” (DNA), used in conjunction with CR0 bit 3, the “Task Switched” bit (TS). (See “Control Registers” in Chapter 2 of the *Intel Architecture Software Developer’s Manual, Volume 3*.) Every task switch via the hardware supported task switching mechanism (see “Task Switching” in Chapter 6 of the *Intel Architecture Software Developer’s Manual, Volume 3*) sets TS. Multi-threaded kernels that use software task switching[1] can set the TS bit by reading CR0, ORing a “1” into[2] bit 3, and writing back CR0. Any subsequent floating point instructions (now being executed in a new thread context) will fault via interrupt 7 before execution. This allows a DNA handler to save the old floating point context and reload the FPU state for the current thread. The handler should clear the TS bit before exit using the CLTS instruction. On return from the handler the faulting thread will proceed with its floating point computation.

Some operating systems save the FPU context on every task switch, typically because they also change the linear address space between tasks. The problem and solution discussed in the following sections apply to these operating systems also.

1. In a software task switch, the operating system uses a sequence of instructions to save the suspending thread’s state and restore the resuming thread’s state, instead of the single long non-interruptible task switch operation provided by the Intel Architecture.
2. Although CR0, bit 2, the emulation flag (EM), also causes a DNA exception, **do not** use the EM bit as a surrogate for TS. EM means that no floating point unit is available and that floating point instructions must be emulated. Using EM to trap on task switches is not compatible with the Intel Architecture’s MMX™ technology. If the EM flag is set, MMX instructions raise the invalid opcode exception.

35.3.6.2 Tracking FPU Ownership

Since the contents of the FPU may not belong to the currently executing thread, the thread identifier for the last FPU user needs to be tracked separately. This is not complicated; the kernel should simply provide a variable to store the thread identifier of the FPU owner, separate from the variable that stores the identifier for the currently executing thread. This variable is updated in the DNA exception handler, and is used by the DNA exception handler to find the FPU save areas of the old and new threads. A simplified flow for a DNA exception handler is then:

1. Use the "FPU Owner" variable to find the FPU save area of the last thread to use the FPU.
2. Save the FPU contents to the old thread's save area, typically using an FNSAVE instruction.
3. Set the FPU Owner variable to the identify the currently executing thread.
4. Reload the FPU contents from the new thread's save area, typically using an FRSTOR instruction.
5. Clear TS using the CLTS instruction and exit the DNA exception handler.

While this flow covers the basic requirements for speculatively deferred FPU state swaps, there are some additional subtleties that need to be handled in a robust implementation.

35.3.6.3 interaction of FPU State Saves and Floating Point Exception Association

Recall these key points from earlier in this document: When considering floating point exceptions across all implementations of the Intel Architecture, and across all floating point instructions, an floating point exception can be initiated from any time during the excepting floating point instruction, up to just before the next floating point instruction. The "next" floating point instruction may be the FNSAVE used to save the FPU state for a task switch. In the case of "no-wait:" instructions such as FNSAVE, the interrupt from a previously excepting instruction (NE=0 case) may arrive just before the no-wait instruction, during, or shortly thereafter with a system dependent delay. Note that this implies that an floating point exception might be registered during the state swap process itself, and the kernel and floating point exception interrupt handler must be prepared for this case.

A simple way to handle the case of exceptions arriving during FPU state swaps is to allow the kernel to be one of the FPU owning threads. A reserved thread identifier is used to indicate kernel ownership of the FPU. During an floating point state swap, the "FPU owner" variable should be set to indicate the kernel as the current owner. At the completion of the state swap, the variable should be set to indicate the new owning thread. The numeric exception handler needs to check the FPU owner and **discard** any numeric exceptions that occur while the kernel is the FPU owner. A more general flow for a DNA exception handler that handles this case is shown in Figure 35-5.

Numeric exceptions received while the kernel owns the FPU for a state swap must be discarded in the kernel without being dispatched to a handler. A flow for a numeric exception dispatch routine is shown in Figure 35-6.

It may at first glance seem that there is a possibility of floating point exceptions being lost because of exceptions that are discarded during state swaps. This is not the case, as the exception will be re-issued when the floating point state is reloaded. Walking through state swaps both with and without pending numeric exceptions will clarify the operation of these two handlers.

Case #1: FPU State Swap Without Numeric Exception

Assume two threads A and B, both using the floating point unit. Let A be the thread to have most recently executed a floating point instruction, with no pending numeric exceptions. Let B be the currently executing thread. CR0.TS was set when thread A was suspended. When B starts to execute a floating point instruction the instruction will fault with the DNA exception because TS is set.

At this point the handler is entered, and eventually it finds that the current FPU Owner is not the currently executing thread. To guard the FPU state swap from extraneous numeric exceptions, the FPU Owner is set to be the kernel. The old owner's FPU state is saved with FNSAVE, and the current thread's FPU state is restored with FRSTOR. Before exiting, the FPU owner is set to thread B, and the TS bit is cleared.

On exit, thread B resumes execution of the faulting floating point instruction and continues.

Case #2: FPU State Swap with Discarded Numeric Exception

Again, assume two threads A and B, both using the floating point unit. Let A be the thread to have most recently executed a floating point instruction, but this time let there be a pending numeric exception. Let B be the currently executing thread. When B starts to execute a floating point instruction the instruction will fault with the DNA exception and enter the DNA handler. (If both numeric and DNA exceptions are pending, the DNA exception takes precedence, in order to support handling the numeric exception in its own context.)

Figure 35-5. General Program Flow for DNA Exception Handler

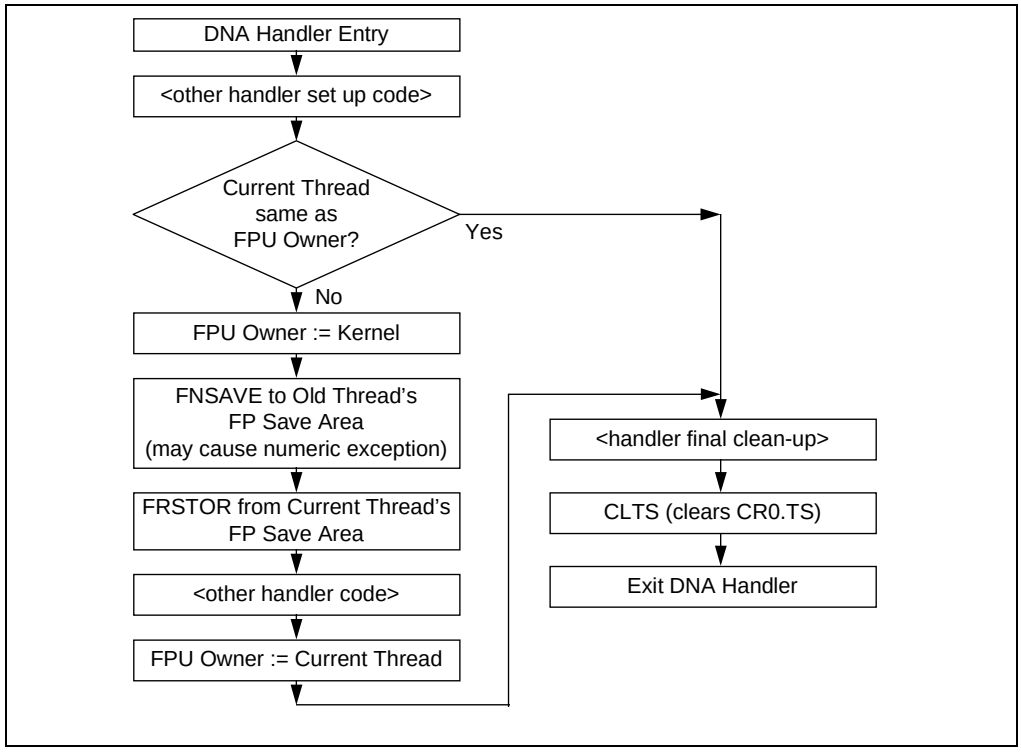
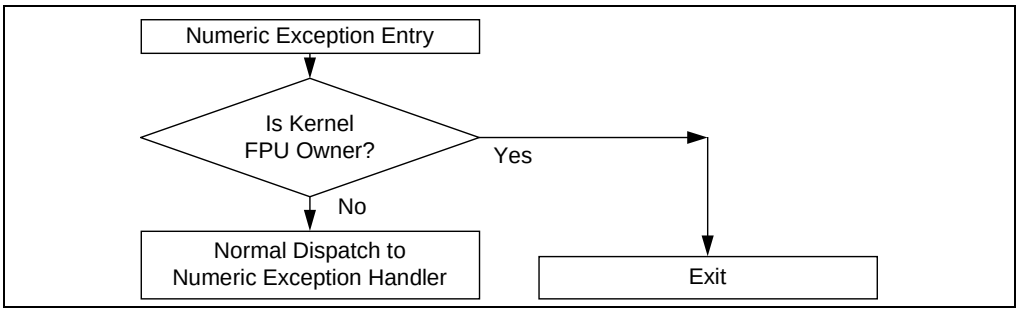


Figure 35-6. Program Flow for a Numeric Exception Dispatch Routine



When the FNSAVE starts, it will trigger an interrupt via FERR# because of the pending numeric exception. After some system dependent delay, the numeric exception handler is entered. It may be entered before the FNSAVE starts to execute, or it may be entered shortly after execution of the FNSAVE. Since the FPU Owner is the kernel, the numeric exception handler simply exits, discarding the exception. The DNA handler resumes execution, completing the FNSAVE of the old floating point context of thread A and the FRSTOR of the floating point context for thread B.

Thread A eventually gets an opportunity to handle the exception that was discarded during the task switch. After some time, thread B is suspended, and thread A resumes execution. When thread A starts to execute an floating point instruction, once again the DNA exception handler is entered. B's FPU state is Finessed, and A's FPU state is Frustrate. Note that in restoring the FPU state from A's save area, the pending numeric exception flags are reloaded in to the floating point status word. Now when the DNA exception handler returns, thread A resumes execution of the faulting floating point instruction just long enough to immediately generate a numeric exception, which now gets handled in the normal way. The net result is that the task switch and resulting FPU state swap via the DNA exception handler causes an extra numeric exception which can be safely discarded.

35.3.6.4 Interrupt Routing From the Kernel

In MS-DOS, an application that wishes to handle numeric exceptions hooks interrupt 16 by placing its handler address in the interrupt vector table, and exiting via a jump to the previous interrupt 16 handler. Protected mode systems that run MS-DOS programs under a subsystem can emulate this exception delivery mechanism. For example, assume a protected mode O.S. that runs with CR.NE = 1, and that runs MS-DOS programs in a virtual machine subsystem. The MS-DOS program is set up in a virtual machine that provides a virtualized interrupt table. The MS-DOS application hooks interrupt 16 in the virtual machine in the normal way. A numeric exception will trap to the kernel via the real INT 16 residing in the kernel at ring 0. The INT 16 handler in the kernel then locates the correct MS-DOS virtual machine, and reflects the interrupt to the virtual machine monitor. The virtual machine monitor then emulates an interrupt by jumping through the address in the virtualized interrupt table, eventually reaching the application's numeric exception handler.

35.4 Differences For Handlers Using Native Mode

The 8087 has a pin INT which it asserts when an unmasked exception occurs. But there is no interrupt input pin in the 8086 or 8088 dedicated to its attachment, nor an interrupt vector number in the 8086 or 8088 specific for an FPU error assertion. But beginning with the Intel 286 and Intel 287 hardware connections were dedicated to support the FPU exception, and interrupt vector 16 assigned to it.

35.4.1 Origin With the Intel 286 and Intel 287, and Intel386™ and Intel 387 Processors

The Intel 286 and Intel 287, and Intel386 and Intel 387 processor/coprocessor pairs are each provided with ERROR# pins that are recommended to be connected between the processor and FPU. If this is done, when an unmasked FPU exception occurs, the FPU records the exception, and asserts its ERROR# pin. The processor recognizes this active condition of the ERROR# status line immediately before execution of the next WAIT or FPU instruction (except for the no-wait type) in its instruction stream, and branches to the routine at interrupt vector 16. Thus an FPU exception will be handled before any other FPU instruction (after the one causing the error) is executed (except for no-wait instructions, which will be executed without triggering the FPU exception interrupt, but it will remain pending).

Using the dedicated interrupt 16 for FPU exception handling is referred to as the native mode. It is the simplest approach, and the one recommended most highly by Intel.

35.4.2 Changes with Intel486™, Pentium® and Pentium Pro Processors with CR0.NE=1

With these latest three generations of the Intel Architecture, more enhancements and speedup features have been added to the corresponding FPUs. Also, the FPU is now built into the same chip as the processor, which allows further increases in the speed at which the FPU can operate as part of the integrated system. This also means that the native mode of FPU exception handling, selected by setting bit NE of register CR0 to 1, is now entirely internal.

If an unmasked exception occurs during an FPU instruction, the FPU records the exception internally, and triggers the exception handler through interrupt 16 immediately before execution of the next WAIT or FPU instruction (except for no-wait instructions, which will be executed as described in “Origin With the Intel 286 and Intel 287, and Intel386™ and Intel 387 Processors”).

An unmasked numerical exception causes the FERR# output to be activated even with NE=1, and at exactly the same point in the program flow as it would have been asserted if NE were zero. However, the system would not connect FERR# to a PIC to generate INTR when operating in the native, internal mode. (If the hardware of a system has FERR# connected to trigger IRQ13 in order to support MS-DOS, but an O/S using the native mode is actually running the system, it is the O/S’s responsibility to make sure that IRQ13 is not enabled in the slave PIC.) With this configuration a system is immune to the problem discussed in “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, where for Intel486 and Pentium processors a no-wait FPU instruction can get an FPU exception.

35.4.3 Considerations When FPU Shared Between Tasks Using Native Mode

The protocols recommended in “Considerations When FPU Shared Between Tasks”, for MS-DOS compatibility FPU exception handlers that are shared between tasks may be used without change with the native mode. However, the protocols for a handler written specifically for native mode can be simplified, because the problem of a spurious floating point exception interrupt occurring while the kernel is executing cannot happen in native mode.

The problem as actually found in practical code in a MS-DOS compatibility system happens when the DNA handler uses FNSAVE to switch FPU contexts. If an FPU exception is active, then FNSAVE triggers FERR# briefly, which usually will cause the FPU exception handler to be invoked inside the DNA handler. In native mode, neither FNSAVE nor any other no-wait instructions can trigger interrupt 16. (As discussed above, FERR# gets asserted independent of the value of the NE bit, but when NE=1, the O/S should not enable its path through the PIC.) Another possible (very rare) way a floating point exception interrupt could occur while the kernel is executing is by an FPU immediate exception case having its interrupt delayed by the external hardware until execution has switched to the kernel. This also cannot happen in native mode because there is no delay through external hardware.

Thus the native mode FPU exception handler can omit the test to see if the kernel is the FPU owner, and the DNA handler for a native mode system can omit the step of setting the kernel as the FPU owner at the handler’s beginning. Since however these simplifications are minor and save little code, it would be a reasonable and conservative habit (as long as the MS-DOS compatibility mode is widely used) to include these steps in all systems.

Note that the special DP (Dual Processing) mode for Pentium Processors, and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors, support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatibility mode for systems using more than one processor.